

# 9

## Driving

In this chapter, we will look at another specialized AI, driving. The other AI we have looked at so far had pretty simple movement for characters. However, car movement needs to take into account physics, and this makes driving AI more complex, which is why we need an AI system specially designed for driving. The AI driving system we will use for our demos is Smart Car AI. Smart Car uses Unity's built-in navigation mesh system, so we will also take a look at it.

In this chapter, you will learn about:

- Setting up the AI driving system
- Creating a Unity navigation mesh
- Using Smart Car to drive AI along a path
- Using Smart Car to drive and avoid obstacles

## An overview of driving

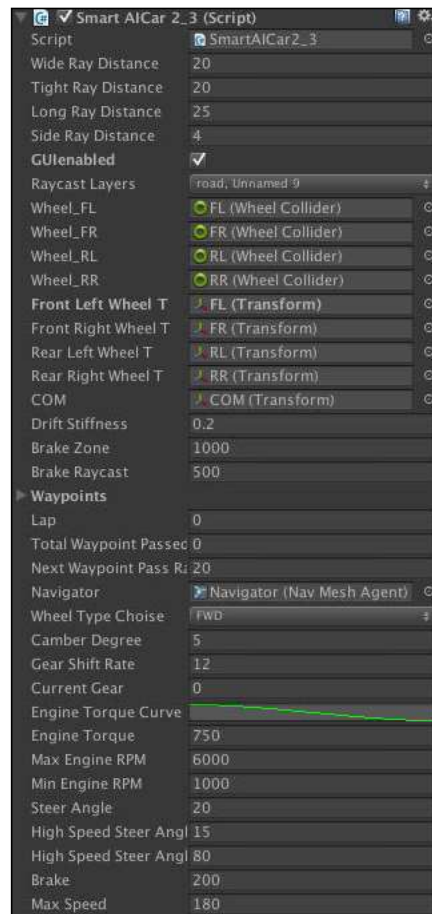
When designing AI for our characters, one of the basic concepts is to have AI move with the same rules as the player. If you ever played any old racing games, sometimes the opponent cars wouldn't follow the same physics as the player, zooming along unrealistically and therefore creating a bad player experience. So, it's important to take car physics into account, including the shape of the car and four wheels, and have the AI move in the same way as the player. This is the main reason for using an AI system especially designed for autos and driving, instead of a general-purpose game AI system we have been using such as RAIN.

The driving system we'll use is Smart Car AI by BoneBreaker, which at the time of writing this book is available in Unity Asset Store for \$10. It takes into account physics for the car and uses ray casting to sense the car's environment. It actually uses two systems for navigation, which are Unity's built-in navigation system to determine paths along a road and ray casting to sense obstacles and make adjustments to the car.

Additionally, Smart Car uses four-wheel physics for realistic movement. Because of the advanced use of physics, we can't just drop any car model in and have it work automatically; we will need to configure Smart Car to use the model's wheel colliders. Wheel colliders are a type of Unity's physics colliders that are specifically made for vehicles. Let's look at how to set up a car.

## Setting up a Smart Car vehicle

As Smart Car uses a realistic car setup, there are many options to configure your vehicle. To create a Smart Car vehicle, you'll need a car model with different models for wheels and Unity wheel colliders setup on them. After adding a car model to your scene, import the Smart Car 2.3 package and attach the `SmartAICar2_3.cs` script from `SmartAICar2.3/Scripts`. In the following screenshot, you can see some of the Smart Car AI fields from the script:



There are fields here you can customize such as Engine properties, including **Engine Torque Curve**, and the distances for the ray casts used for sensing. Most of these can be left to default values, but to run the script, you'll need to fill in the wheel properties, dragging from your model the colliders for the four wheels to the **Wheel\_FL**, **Wheel\_FR**, **Wheel\_RL**, and **Wheel\_RR** properties. You also need transforms for the wheel set. Also, there needs to be a transform for **center of mass (COM)**, a lower point in the middle of the car. If COM is placed in the wrong position, the physics of the car can be very unexpected. If you fill these out, the car is set up but it still won't run in a game as it still needs waypoints and a Unity navigation mesh setup, which we will add in the demo.

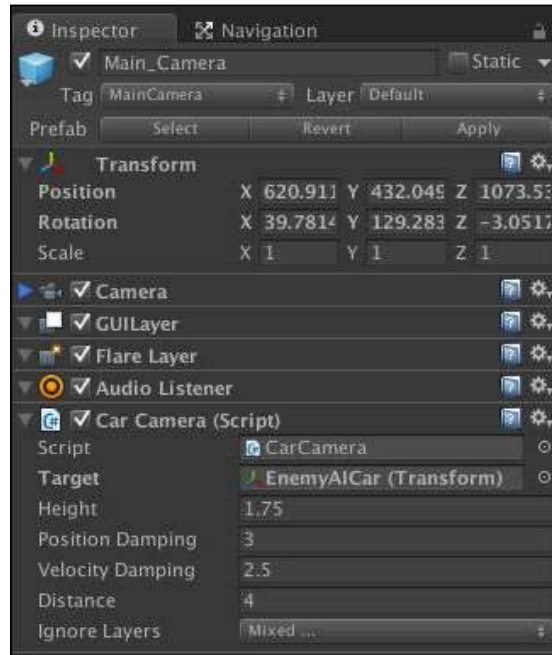
## The Smart Car AI demo

Now, we'll start setting up our driving demo that will have a car driving along a road and avoiding obstacles.


## Setting up a Unity test scene

Besides needing Smart Car, we'll need an environment for our AI cars to drive in. We'll use Car Tutorial v1.3 that is made by Unity, which you can download for free from the Asset Store. Import the project and open **TheTrack** scene from the imported Scenes folder. Next, add a car to the scene. The car prefab that comes with Car Tutorial doesn't have the complete wheel physics setup, so you can configure it using the steps in the last section or use the **EnemyAICar** prefab from Smart Car. To make the car work better with the **Tutorial** scene, extend the rays a little, set **Wide** and **Tight Ray Distance** both to 40 and **Long Ray Distance** to 50. This keeps the car from hitting obstacles when going too fast and missing tight turns. Once you have a car in the scene configured for Smart Car, select the **Main\_Camera** object and set your car to **Target** for the **Car Camera** script.

If you start the demo now, the car still won't run but the main camera in the scene will follow it:



This screenshot is of the **Main\_Camera** game object of Car Tutorial. These are the settings for the **TheTrack** scene with **Target** set to the Smart Car prefab.

[  Another setting in the scene that can cause problems is the TunnelSoundTrigger Sound Toggler script. As it isn't important to use, select the TunnelSoundTrigger script and remove that component to avoid errors later. ]

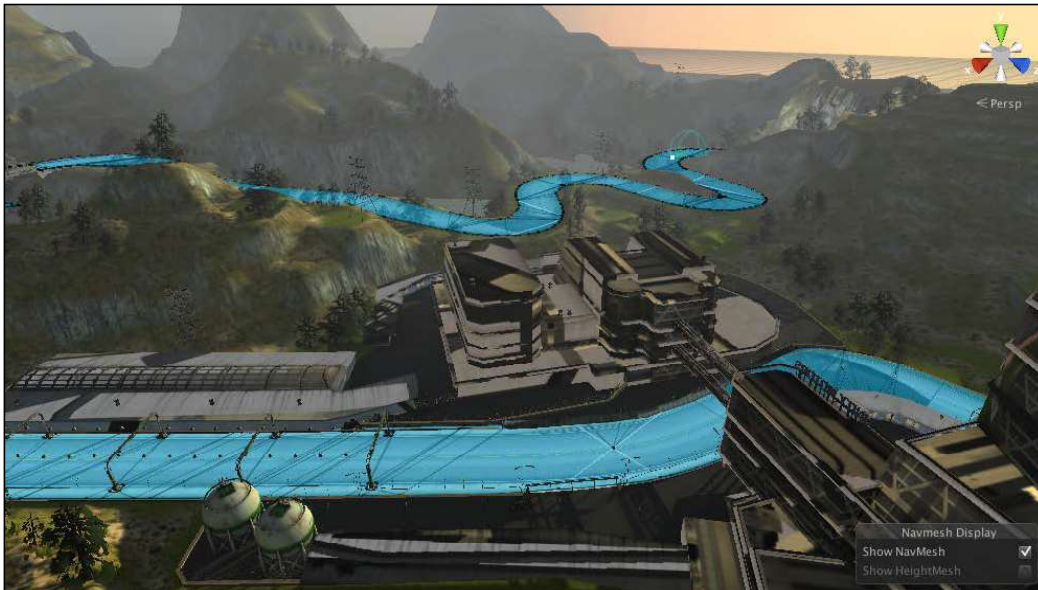
## Using Unity's built-in NavMesh system

The next thing we need for our car demo is a navigation mesh. Smart Car uses Unity's built-in system. Unity's system is similar to RAIN's but we haven't used it much yet as unlike other plugins, Unity does not have a built-in behavior tree system. Fortunately, we don't need behavior trees for our car demos, so navigate to **Window | Navigation**.

This brings up the **Navigation** tab with three subtabs to help configure the NavMesh:

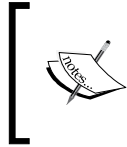
- **Object:** This helps you filter what objects in the scene are part of the navigation mesh. Any objects that are tagged with **Navigation Static** will be included in the mesh as a walkable area.
- **Bake:** This has options to bake the mesh. The two most important options are **Radius** and **Height**, which are dimensions for the character to navigate on the mesh.
- **Layers:** This allows you to customize the placement of different navigation meshes on different layers.

For our demo, we want only the roads to be navigable. Select the other building and miscellaneous objects in the scene and set their static property (which is to the right of their name in **Inspector**) to not have **Navigation Static** set. Then, for the different road objects, such as **Road\_Coll**, **Road\_Coll01**, and so on, make sure that they have **Navigation Static** checked. Then, go back to the **Navigation** tab and click on **Bake**. If you have everything set correctly after you bake, you should see the navigation mesh in the same area as the road:



This is how the road navigation mesh setup should look.

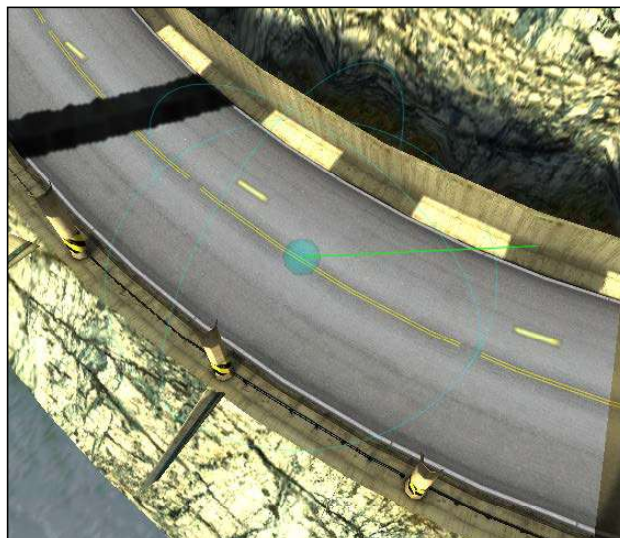
This should have been pretty quick to recreate, but depending on the character size settings and the amount of geometry in the scene, this can take a bit of time. We will discuss navigation meshes more and the algorithm behind how they are generated in *Chapter 11, Advanced NavMesh Generation*.



**NavMeshAgent** is the built-in Unity component to create characters that move on a navigation mesh. Smart Car uses this internally. We won't be using this class directly but you can if you want to try more of Unity's built-in navigation system; it is a good class to look at.

## Setting up waypoints

The final step to get a car driving is to set up waypoints for the car to follow. The NavMesh we created defines the area that the car can navigate to and the waypoints define the general path the car should follow. Create a new empty game object and name it **Road Waypoints**. Then, create a few more empties with the names **waypoint 1**, **waypoint 2**, **waypoint 3**, and so on. Place the waypoint empties at different parts along the road. Note that the NavMesh for the road will define how to get from one waypoint to the next, so the line between waypoints doesn't have to go through the road. For instance, you could have one waypoint at the start of a curve and the second at the end and the car would still go around the curve through the waypoints. In the Smart AICar script, set the empties to the **Waypoints** field. After doing this, the waypoints will be visualized in the edit or view to make adjusting their locations easier. Refer to the following screenshot, and you can see how the visualization of Smart car AI waypoints looks:





If you run the demo after setting the waypoints, the car drives realistically across the road. If you want to fine-tune the car more, remember there are many physics settings with Smart Car that can be adjusted to change how the car acts.

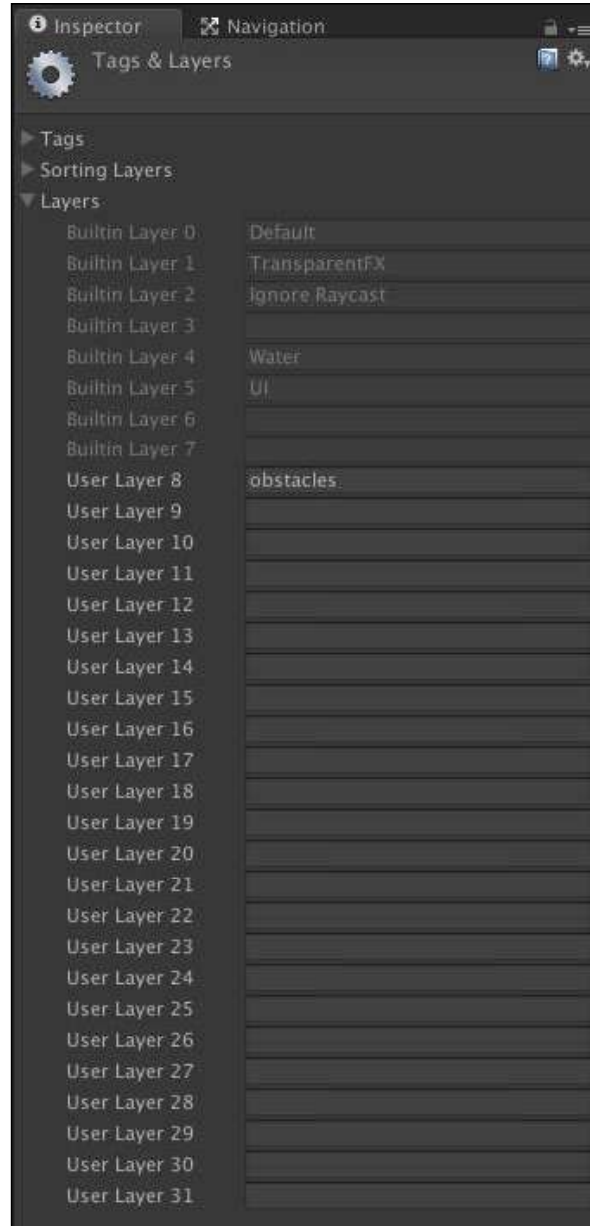
## Adding obstacles to driving

As Smart Car uses a combination of a NavMesh and ray casting, you can add objects dynamically to the scene, and as long as they have colliders attached (and are on a car's **Recast Layers**), the car will avoid them. To try this out, add a few large cylinders to the road, as shown in the following screenshot:



Then, in the **Recast Layers** dropdown for your car, make sure that it is set to ray cast on the same layer as the obstacles. Select a **Cylinder** object and in **Inspector**, select **Add Layer**. We need to create an obstacles layer, so select the dropdown and in the first slot for **User Layer**, set it to **obstacles**.

Then, for each cylinder, set its layer to **obstacles**:





This is how the **Inspector** window should look after creating the obstacle layer.

Then, for Smart Car in the **Raycast Layers** dropdown, make sure that the **obstacles** layer is selected. Once you have this set up, if you run the demo, the car will drive and avoid the obstacles. The car still uses physics for its control, and sometimes if you place the obstacles too close to one another, the car will run into one. Fortunately, in this case, the car will back up and then drive past it, which is a nice touch Smart Car has.

The cylinders are static objects in our scene, but as ray casting is used, there is no reason why you cannot script dynamic objects and the car will still avoid them. To see this, run the demo and in the scene view, grab one of the obstacles and move it around to block the car; the car will try to avoid it.

## Additional features

We've just completed creating a driving demo with a car avoiding obstacles, but there are a few more things you can do with driving AI. We can add brake and drift zones to help configure the general behavior of the car as it drives around the scene, and we can integrate Smart Car with other AI systems such as RAIN.

## Adding brake zones and drift zones

Another interesting thing you can do with Smart Car is define zones in the level to either cause the car to brake and slow down or adjust the friction of the car to make it drift. These are similar to the vector fields we saw in *Chapter 5, Crowd Control*, where we place them in the level to affect the AI, and they aren't visible to the player but are good to use for scripting level experiences. To create a brake or drift zone in your game, add a cube to the game (go to **GameObject | Create Other | Cube**) and scale and translate the area you want to tag in the level. In the Inspector window for the cube, set its tag to **BrakeZone** and for a drift zone set the tag to **DriftZone**. Next, in **Box Collider** for the cube, check **Is Trigger** to true, so the car will get a message of intersecting with a cube but won't stop and collide with it. Lastly, in the **Inspector** window, uncheck **Mesh Renderer** so that the cube is invisible in the game. Now when you run the demo, if the car's speed is **25** or over when it enters the brake zone, you will see it slow down, and if its speed is **15** or over, you will see it drift in the drift zone.

## Integrating with other AI systems

In this demo, we've seen that setting up an AI car that drives around is easy to do with Smart Car. However, what if your game isn't just a driving game but has car driving as one part of the game? If that's the case, you can mix Smart Car with another AI system easily. For RAIN integration, import the RAIN package into your scene. Then, go to **RAIN | Create Entity** and then select **Add Aspect: Visual Aspects**. This creates an entity with an aspect that can be sensed by additional RAIN AI entities you can create in the scene, making the car just one part of a larger AI system.

## Summary

In this chapter, we looked at Smart Car, an AI system specifically for car AI. We discussed why automotive AI is different than most AIs because of the physics involved, and we also saw how to set up a car model, create a path for the car, and add obstacles. We also looked at using Unity's built-in navigation mesh system, instead of using third-party ones such as RAIN, and discussed additional features for car AI and how we can integrate it with another AI system such as RAIN.

In the next two chapters, we will look at how to combine character animations and AI to give them a realistic appearance and learn more about creating complex navigation meshes for different AIs.